# Relational Models for a Language with Effect Capabilities

## (Efektodziej i łapacz, czyli modele języka programowania z efektami algebraicznymi)

Patrycja Balik

Praca magisterska

**Promotor:** Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

26 sierpnia 2024

# Abstract

In the world of programming languages, there exist several different approaches to algebraic effects and their handlers. This work focuses on one such approach, known as the capability-passing style. In this style every handler binds what is called an effect capability, which is a regular first-class value that can be used to invoke the effect associated with the handler. This technique offers us a lot of flexibility in the shape of this capability by reusing existing language mechanisms, such as records, polymorphism and more. Our goal is to define logical relations for a calculus with effect capabilities. Previous work in this area did not account for the full flexibility afforded by capabilities, instead opting to limit them to a single syntactic form. To fill this gap, we propose two different relational models that scale well to a calculus that features multiple shapes of capabilities that are useful in practice.

---

W świecie języków programowania istnieje kilka różnych podejść do efektów algebraicznych i ich łapaczy (ang. *handler*). Ta praca skupia się na jednym z nich, zwanym stylem przekazywania efektodziejów (ang. *capability*). Cechą charakterystyczną tego stylu jest to, że każdy łapacz wiąże zwykłą wartość pierwszej kategorii zwaną efektodziejem, którego używa się do wykonania efektu powiązanego z tym łapaczem. Ta technika oferuje dużą elastyczność co do kształtu efektodzieja poprzez wykorzystanie istniejących mechanizmów języka takich jak rekordy, polimorfizm i inne. Naszym celem jest zdefiniowanie relacji logicznych dla rachunku z efektodziejami. Istniejące wyniki w tej dziedzinie nie wykorzystywały w pełni elastyczności, na którą pozwalają efektodzieje, lecz ograniczały je do pojedynczej formy syntaktycznej. Aby wypełnić tę lukę, proponujemy dwa różne modele relacyjne, które dobrze skalują się na rachunek z wieloma różnymi formami efektodziejów, które są przydatne w praktyce.

# Contents

# 1 Introduction

Algebraic effects and handlers [Plotkin and Pretnar 2013] have spurred a wealth of research too broad to recount here in full,[1] and are enjoying great popularity in a practical setting as well, be that in the many experimental programming languages [Biernacki *et al.* 2019; Brachthäuser *et al.* 2020a; Hillerström and Lindley 2016; Leijen 2014; Lindley *et al.* 2017], or well-established ones, such as OCaml [Sivaramakrishnan *et al.* 2021]. As a result, there are several different approaches to handlers, each with a different set of advantages and trade-offs. This thesis will focus on one particular point in this design space, often called the capability-passing style [Brachthäuser *et al.* 2020a,b], which itself is a form of lexical effect handlers [Biernacki *et al.* 2020; Zhang and Myers 2019]. We will describe how capability-passing works from the programmer's point of view in Chapter 2.

The calculus studied in this work is motivated by practical needs, as it was designed to model a simplified version of the Fram programming language,[2] which is being developed at the Institute of Computer Science, University of Wrocław. A large part of the goal of this thesis is to provide a solid theoretical foundation for the effect handlers in Fram by defining logical relations that can adequately support all desired language features. This motivation stems from the belief that an elegant relational model leads to better language design, with clear, intuitive behavior and well-defined static guarantees. In fact, the process of developing the theory presented in this work influenced the design of handlers in Fram.

Logical relations as a tool have many applications, such as proving properties of a language like termination or type safety, or characterizing and reasoning about contextual equivalence. While our logical relations do entail contextual approximation, we are much more interested in the construction of the relational model itself, since it can be a source of insight into the language's semantics in its own right.

Biernacki *et al.* [2018] introduced a technique for constructing a relational model for algebraic effects with dynamic (non-lexical) semantics. Later on, that approach was used by Zhang and Myers [2019] and Biernacki *et al.* [2020] to develop models for lexical handlers. Of those two works, the one due to Zhang and Myers featured capability-passing style. However, their calculus lacked useful features like effects with multiple operations and polymorphic operations, and their relational model cannot support those and other additions without modifications. In this work, we propose two different models that can easily scale to many shapes of handlers without any kind of tedious case analysis.

Despite the theoretical nature of this work, theorems will be stated without presenting their proofs in the text. This is because both of the proposed relational models are fully formalized using the Coq proof assistant. Though the upfront effort required for proof mechanization is significant, automated proof checking not only makes the results more trustworthy, but also eases the process of later modification of the definitions and allows

[1] For a taste of it, see https://github.com/yallop/effects-bibliography.

[2] Fram can be found at https://github.com/fram-lang/dbl.

for proof reuse. In the spirit of reuse, the formalization of this work relies on two third party libraries, Binding [Polesiuk and Sieczkowski 2024] and IxFree [Polesiuk 2017].

The technical material contained in this thesis is the result of the joint work of its author and the advisor, Piotr Polesiuk. The results included herein will be presented as part of a talk at the HOPE 2024 workshop [Balik and Polesiuk 2024].

# 2 Programming with Algebraic Effects

To start off, we will go over small programming examples to illustrate some aspects of programming with algebraic effects and handlers and to motivate the choices made in the technical portion of this work. Insofar as it is possible, in this chapter we will use syntax similar to that of Fram, especially in Section 2.1, which introduces effect capabilities.

The key feature of algebraic effects is the ability to locally define the semantics of user-defined effects by using the **`handle ... in ...`** construct. For a very simple example, we can consider the reader effect with its single operation `ask`, which takes a unit argument and returns a single integer value. When an effectful operation is invoked, the current context up to and including the **`handle`** construct is captured into a *resumption*, and the body of the operation's definition is evaluated. In the program below, `ask` binds the resumption to the variable `resume`, and immediately calls it with the argument `21`. As a result, each call to `ask` in the handled expression evaluates to `21`.

```
handle ask () / resume => resume 21 in ask () + ask ()
```

A more interesting example of an effect is a single cell of mutable integer state. At its most basic, state requires two operations: `get`, which takes a unit argument and returns the current value of state, and `put`, which sets the state to the integer received as its argument and returns unit. In the following, we implement a particular handler for state with a hardcoded initial value of `13`.

```
handle
  get () / resume => fn s => resume s  s
  set s' / resume => fn _ => resume () s'
  return  x => fn _ => x
  finally f => f 13
in
let x = get () in
set 29; x + get ()
```

Stateful computations are represented using functions that accept the current value of state. For example, `get` is implemented as a function that resumes the computation as `resume s`, where `s` is being returned to the operation's caller, and runs the whole thing with the unchanged state by passing `s` again. The operation `put` ignores the current state, resumes with the unit value, and runs with the new state received as an argument. The **`return`** clause describes what to do if the computation under **`handle`** evaluates to a value. In this case, we use it to turn the value into a constant function ignoring the current state, because the type of the computation has to be consistent whether we return or perform an operation. At last, the **`finally`** clause as used here behaves a lot as if it was syntactic sugar, where **`handle`** $h$ **`finally`** `x =>` $e_f$ **`in`** $e$ stands for **`let`** `x =` `(`**`handle`** $h$ **`in`** $e$`)` **`in`** $e_f$. We use it to pass the initial value of state to the computation.

Contrary to the examples that we have seen so far, we do not have to write all of the invocations of an effect syntactically within its `handle ... in ...` expression. To illustrate this, we can refactor the first example by moving the expression under the `handle` construct into a separate function.

```
let ask_twice () = ask () + ask () in

handle ask () / resume => resume 21 in ask_twice ()
```

We may well also use multiple instances of `handle` for the same effect, even nested within one another, as in the following example.

```
handle
  ask () / resume => resume 10
in
ask_twice () + (handle ask () / resume => resume 11 in ask_twice ())
```

Which handler any given call to `ask_twice` refers to clearly influences the result. Originally, algebraic effect handlers were *dynamic*: the dynamically nearest handler of an effect is the one to use for each use of an effectful operation. This is quite similar to the semantics of exception handlers in many programming languages. With this semantics, the left-most call to `ask_twice` refers to the outer handler, which is the only one in dynamic scope at this point, and returns `20`. The other call uses the innermost handler, returning `22`.

One disadvantage of this semantics is that it makes it difficult to intentionally refer to any handler other than the closest at runtime. Solutions based on coercions [Biernacki *et al.* 2019] or adaptors [Convent *et al.* 2020] are unwieldy to use due to forcing the programmer to count the number of intervening handlers, which hurts readability and maintainability.

The other issue with dynamic handlers is that they are susceptible to accidental effect capture, where the use of an effect is unintentionally intercepted by a different handler than the programmer intended. We borrow the classic example of this from Biernacki *et al.* [2018]. First, we define a function `count`, which counts the number of times the higher-order function `f` calls its argument `g`. To do this, we use a counter effect with a single operation `tick : Unit -> Unit` and define a handler for it in a similar manner to the state handler that we saw earlier.

```
let count f g =
  handle
    tick () / resume => fn n => resume () (n + 1)
    return _         => fn n => n
    finally f        => f 0
  in
  f (fn x => tick (); g x)
```

With the dynamic semantics of handlers, we run into problems if the functions passed to `count` do something with `tick` themselves. For example, we can pass `count` to itself as the first argument: `count count (fn x => x)`. Even though the inner `count` calls the function it receives as an argument, the inner handler intercepts the invocation of `tick` inserted into the argument by the outer `count`, so the entire expression evaluates to 0.

## 2.1 Lexical Handlers

*Lexical handlers* [Biernacki *et al.* 2020; Zhang and Myers 2019] provide an alternative to dynamic handlers which does not suffer from the mentioned problems. The main idea is that each **handle** expression binds a new variable, representing an effect instance. In the simplest form, the programmer needs to explicitly associate the use of an effectful operation with its handler by using such a variable. To connect an operation with its handler across function calls, the variable can be abstracted by and passed to functions.

There are a few different approaches to lexical handlers. One choice to be made is whether the variable bound by the **handle** construct is first-class [Xie *et al.* 2022] or second-class [Biernacki *et al.* 2020; Zhang and Myers 2019]. The second decision is whether effects and their operations need to be declared upfront, or if **handle** binds the variable to a regular value, such as a function, with no dedicated notion of operations. The latter approach is known as the *capability-passing style* [Brachthäuser *et al.* 2020a,b], while the value defined by **handle** is called a *capability*. In this work, we consider first-class effect instances in the capability-passing style.

The astute reader might have noted that in case of the reader example at the beginning of this chapter, we hardly needed **handle** at all: a simple let-definition would do, as follows.

```
let ask = fn () => 21 in
ask () + ask ()
```

In this case, creating multiple instances of `ask` is as simple as defining multiple functions with distinct identifiers. These can be easily passed to other functions as well.

```
let add ask_a ask_b = ask_a () + ask_b () in

let ask_a = fn () => 29 in
let ask_b = fn () => 13 in
add ask_a ask_b
```

While not all effects can be easily expressed in this way, in capability-passing style, we can allow **handle** to look a lot like **let**.

```
handle ask_a = effect () / resume => resume 29 in
handle ask_b = effect () / resume => resume 13 in
add ask_a ask_b
```

The difference is that **handle** sets up a delimiter for the special control flow of the newly-created effect, and can use the **effect** construct to define an effectful function that has access to the continuation `resume`. As we will describe later in this chapter, the type system additionally ensures that the capabilities are never called outside of their respective handlers. Note that, just like when we used **let** and regular lambdas, `ask_a` and `ask_b` are both functions, and can be passed as arguments to `add`.

We are not restricted to immediately using **effect** in a **handle** expression, either. To define a handler for an effect with multiple operations, such as state, we can use a pair or record of multiple effectful functions as in the following example.

```
handle st = State
  { get = effect () / r => fn s => r s  s
  , set = effect s  / r => fn _ => r () s
  }
  return x  => fn _ => x
  finally f => f 0
in
st.set 21; st.get () + st.get ()
```

Note that as before, we can include the **return** and **finally** clauses.

Besides value constructors, it may be convenient to allow let-definitions within a handler definition. For example, suppose we want to extend the state effect with an `update` operation, which takes a function that is used to transform the current value o state. One option is to use **effect** to define the new field, just like we did for the other two operations.

```
handle st = State
  { get    = effect () / r => fn s => r s  s
  , set    = effect s  / r => fn _ => r () s
  , update = effect f  / r => fn s => r () (f s)
  }
in ...
```

However, we can instead use the fact that `update` can be easily expressed as a regular function in terms of `get` and `set`. This can be done easily by using **let**.

```
handle st =
  let get = effect () / r => fn s => r s  s in
  let set = effect s  / r => fn _ => r () s in
  let update f = put (f (get ())) in
  State { get, set, update }
in ...
```

### Type and Effect System

One of the appeals of algebraic effects is that they pair well with a type and effect system, which allows us to statically track which computations perform specific effects, and to make sure each effect gets handled. In our setup, each function type is of the shape $\tau_1$ `->[`$\varepsilon$`]` $\tau_2$, containing an argument type $\tau_1$, result type $\tau_2$, and the effect $\varepsilon$ that might occur upon calling the function.

In order to ensure that each effect is always used under the correct handler, the type of functions defined using **effect** is annotated with an effect variable bound by the **handle** construct. Just like regular type variables, the type and effect system ensures that effect variables never escape their scope. This technique has previously been applied to algebraic effects by Xie *et al.* [2022], and is very similar to how Haskell's ST monad utilizes rank-2 types to guarantee safety.

For clarity, we can explicitly annotate the reader example with the binding occurrence `{effect=E}`, naming the bound effect `E`. Since we gave the effect a programmer-facing name, we can also annotate the capability `ask` with its complete type `Unit ->[E] Int`.

```
handle {effect=E} (ask : Unit ->[E] Int) =
  effect () / resume => resume 21
in
  ask () + ask ()
```

Regardless of the annotations present, the typechecker must reject any program that attempts to smuggle the effect `E` out of the scope of **`handle`**, for example one that simply returns the capability uncalled.

```
handle ask = effect () / resume => resume 21 in ask
```

### First-Class Handlers

In some examples discussed thus far, as well as in practice, the same or similar handler definitions are used repeatedly in multiple **`handle`** `x =` *h* **`in`** *e* expressions. Each use of **`effect`** is closely connected to a specific **`handle`**, so we cannot directly move the definition *h* elsewhere for reuse. A possible workaround is to define a higher-order function that takes an effectful computation as an argument. For example, below we implement such a function for the state handler from earlier.

```
let hState (c : {effect=E} -> State E -> _) =
  handle st = State
    { get = effect () / r => fn s => r s  s
    , set = effect s  / r => fn _ => r () s
    }
    return x  => fn _ => x
    finally f => f 0
  in
  c st

let x =
  hState (fn st =>
  st.set 21; st.get () + st.get ())
```

The computation `c` has to be polymorphic in the effect variable associated with **`handle`**, and is given the capability to perform the effect as an argument. While this solution works, it is the source of a fair bit of syntactic overhead for what is a very common pattern.

As it turns out, several programming languages with algebraic effects implement some form first-class handlers (*e.g.*, Koka [Leijen 2014], Helium [Biernacki *et al.* 2019], or OCaml [Sivaramakrishnan *et al.* 2021]), which make handler reuse more convenient. In our case, first-class handlers are values created by using the **`handler`** *h* syntax, where the shape of *h* follows the same rules as in a **`handle`** `x =` *h* definition. A first-class handler *v*

can be installed using `handle x with v`. The previous example can be rewritten using first-class handlers as follows.

```
let hState = handler State
  { get = effect () / r => fn s => r s  s
  , set = effect s  / r => fn _ => r () s
  }
  return x  => fn _ => x
  finally f => f 0

let x =
  handle st with hState in
  st.set 21; st.get () + st.get ()
```

In fact, the syntax with the equals sign can be considered syntactic sugar for immediately installing a handler value: `handle x = h` $\triangleq$ `handle x with handler h`.

First-class handlers have somewhat complicated types, though they closely mirror the types of higher-order functions used as a way to abstract handlers. The handler type `handler E. Th @ Ti / Ei => To / Eo` contains the bound effect variable `E`, the capability type `Th`, the (input) type `Ti` and effect `Ei` of the handled expression, and the (output) type `To` and effect `Eo` of the entire `handle x with ... in ...` expressions. The effect variable can appear in `Th`, `Ti`, and `Ei`. As a handler using the variable in either `To` or `Eo` would be unusable due to violating scoping, it is not allowed to appear there in the first place to catch the mistake earlier. The handled expression is always allowed to perform `E`, even if `Ei` does not explicitly mention it. If we used a handling function instead of a first-class handler, the analogous type would be `({effect=E} -> Th ->[E, Ei] Ti) ->[Eo] To`.

# 3 The Minimal Calculus

In this chapter we will present a minimal calculus with effect capabilities. Many of the examples shown in the previous chapter cannot be written without proper extensions. As these extensions significantly impact the relational model of handlers, we will introduce them gradually in Chapter 4, along with the descriptions of the proposed models.

## 3.1 Syntax

The syntax of the calculus is presented in Figure 3.1. Kinds are used to distinguish types (T) and effects (E). The types of the minimal calculus consist of type variables $\alpha^{\mathsf{T}}$, the unit type 1, effect-annotated arrow types $\tau \to_{\varepsilon} \tau$, and handler types $\mathsf{H}\alpha^{\mathsf{E}}.\, \tau \mathbin{@} \tau \mathbin{/} \varepsilon \Rightarrow \tau \mathbin{/} \varepsilon$. As before, a handler type contains the type of the capability, the handled expression's type and effect, and the overall type and effect of the result. It also binds an effect variable that can be used in the type of the capability, and the type and effect of the handled expression. Effects are either effect variables $\alpha^{\mathsf{E}}$, the pure effect $\iota$, or the join of two effects $\varepsilon \cdot \varepsilon$. The effects are treated like sets, with pure being empty and join acting as union. We will often omit the kind annotation of type and effect variables whenever it is unambiguous from context.

The values of the calculus are variables, the unit value $\langle\rangle$, lambda abstractions $\lambda x.\, e$, and handler values $\texttt{handler}\ h\ \texttt{ret}\ x \Rightarrow e$, which contain a return clause that is assumed to be the identity if omitted. To make the technical presentation a bit simpler, we did not include the finally clause from Chapter 2. Another simplification is that handlers are created using the separate syntactic category of handlers, rather than general expressions. For

$$
\begin{aligned}
\text{Kind} \ni \kappa &::= \mathsf{T} \mid \mathsf{E} && \text{(kinds)}\\
\text{TVar} \ni \alpha^{\kappa}, \beta^{\kappa}, \ldots & && \text{(type variables)}\\
\text{Type} \ni \tau &::= \alpha^{\mathsf{T}} \mid 1 \mid \tau \to_{\varepsilon} \tau \mid \mathsf{H}\alpha^{\mathsf{E}}.\, \tau \mathbin{@} \tau \mathbin{/} \varepsilon \Rightarrow \tau \mathbin{/} \varepsilon \mid \ldots && \text{(types)}\\
\text{Effect} \ni \varepsilon &::= \alpha^{\mathsf{E}} \mid \iota \mid \varepsilon \cdot \varepsilon && \text{(effects)}\\
\\
\text{Var} \ni x, y, f, r, \ldots & && \text{(variables)}\\
\text{Expr} \ni e &::= v \mid e\ e \mid \texttt{handle}\ x\ \texttt{with}\ v\ \texttt{in}\ e \mid \ldots && \text{(expressions)}\\
\text{Val} \ni v &::= x \mid \langle\rangle \mid \lambda x.\, e \mid \texttt{handler}\ h\ \texttt{ret}\ x \Rightarrow e \mid \ldots && \text{(values)}\\
\text{Handler} \ni h &::= \texttt{eff}\ x/r.\, e \mid \ldots && \text{(handlers)}
\end{aligned}
$$

Figure 3.1: The syntax of the minimal calculus of effect handlers

now, it only includes effectful functions $\texttt{eff}\ x/r.\, e$, though we will extend it with multiple additional constructs in Chapter 4, and discuss their impact on the relational model.

The non-value expressions are the standard function application $e\ \ e$ and handling expressions $\texttt{handle}\ x\ \texttt{with}\ v\ \texttt{in}\ e$. As before, the syntax $\texttt{handle}\ x = h\ \texttt{ret}\ x \Rightarrow e_r\ \texttt{in}\ e$ is syntactic sugar for $\texttt{handle}\ x\ \texttt{with}\ \texttt{handler}\ h\ \texttt{ret}\ x \Rightarrow e_r\ \texttt{in}\ e$.

## 3.2 Type System

The typing rules of the calculus are displayed in Figure 3.2. Most of the typing rules for expressions and values are standard, and make use of two environments, $\Delta$ for the type variables, and $\Gamma$ for regular term variables. As values are pure, they are related with just a type, while expressions are associated with a type and an effect. The rule for handling expressions ensures that the handler is of the handler type $\mathsf{H}\alpha.\, \tau_h\ @\ \tau_i\ /\ \varepsilon_i \Rightarrow \tau_o\ /\ \varepsilon_o$. In an environment extended with a new abstract effect $\alpha$ and the capability of type $\tau_h$, the handled expression is of type $\tau_i$ and effect $\alpha \cdot \varepsilon_i$, matching the handler. The entire `handle` expression is similarly of the output type $\tau_o$ and effect $\varepsilon_o$.

The typing rule for handler values uses an auxiliary relation to type the bare handler without its return clause. The relation $\Delta_1; \alpha; \Delta_2; \Gamma \vdash h : \tau_h\ @\ \tau\ /\ \varepsilon$ has a type variable environment $\Delta_1$, a distinguished effect variable $\alpha$, corresponding to this handler's effect, and a second type variable environment $\Delta_2$, containing all type variables bound later than $\alpha$. We will encounter a non-empty $\Delta_2$ once we extend the calculus with universal types. While we could have merged $\Delta_1$ and $\Delta_2$, leading to an equivalent type system, their separation is somewhat more hygienic and more closely matches the construction of one of the relational models. Like the handler type, the handler typing relation contains the capability type $\tau_h$. Finally, $\tau$ and $\varepsilon$ are the type and effect of the delimiter, which are the type and effect produced by the return clause and used as the effect and return type of resumptions. As there is no finally clause, the delimiter type and effect are always, respectively, a subtype and subeffect of the output type and effect in the handler type.

The rule for $\texttt{eff}\ x/r.\, e$ tells us that effectful functions are of the arrow type, annotated with the distinguished effect $\alpha$. The body is checked in an environment with the argument $x$ and resumption $r$.

The type system features subtyping and subeffecting, so each judgement comes with a subsumption rule. Subsumption for values and expressions is standard. The subsumption rule for the handler relation is covariant in the capability type, and invariant in the type and effect of the delimiter. The invariance results from the fact that, in the rule for `effect`, the delimiter type and effect are also used in a negative position in the type of the resumption.

The subeffecting relation $\varepsilon_1 <: \varepsilon_2$ simply encodes the subset-like properties we expect from the relation. Subtyping is largely standard, besides the rule for handler types. This rule is covariant in the capability type and output type and effect, and contravariant in the input type and effect. The effect $\alpha$ may also always be added to the input effect.

$$\frac{\tau_h <: \tau_h' \qquad \tau_i' <: \tau_i \qquad \varepsilon_i' <: \alpha \cdot \varepsilon_i \qquad \tau_o <: \tau_o' \qquad \varepsilon_o <: \varepsilon_o'}{\mathsf{H}\alpha.\, \tau_h\ @\ \tau_i\ /\ \varepsilon_i \Rightarrow \tau_o\ /\ \varepsilon_o <: \mathsf{H}\alpha.\, \tau_h'\ @\ \tau_i'\ /\ \varepsilon_i' \Rightarrow \tau_o'\ /\ \varepsilon_o'}$$

**Typing expressions.** $\boxed{\Delta; \Gamma \vdash e : \tau \;/\; \varepsilon}$

$$\frac{\Delta; \Gamma \vdash v : \tau}{\Delta; \Gamma \vdash v : \tau \;/\; \iota} \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to_\varepsilon \tau_1 \;/\; \varepsilon \qquad \Delta; \Gamma \vdash e_2 : \tau_2 \;/\; \varepsilon}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_1 \;/\; \varepsilon}$$

$$\frac{\Delta; \Gamma \vdash v : \mathsf{H}\alpha.\, \tau_h \mathbin{@} \tau_i \;/\; \varepsilon_i \Rightarrow \tau_o \;/\; \varepsilon_o \qquad \Delta, \alpha; \Gamma, x : \tau_h \vdash e : \tau_i \;/\; \alpha \cdot \varepsilon_i}{\Delta; \Gamma \vdash \mathtt{handle}\ x\ \mathtt{with}\ v\ \mathtt{in}\ e : \tau_o \;/\; \varepsilon_o}$$

$$\frac{\Delta; \Gamma \vdash e : \tau \;/\; \varepsilon \qquad \tau <: \tau' \qquad \varepsilon <: \varepsilon'}{\Delta; \Gamma \vdash e : \tau' \;/\; \varepsilon'}$$

**Typing values.** $\boxed{\Delta; \Gamma \vdash v : \tau}$

$$\frac{(x : \tau) \in \Gamma}{\Delta; \Gamma \vdash x : \tau} \qquad \Delta; \Gamma \vdash \langle\rangle : 1 \qquad \frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \;/\; \varepsilon}{\Delta; \Gamma \vdash \lambda x.\, e : \tau_1 \to_\varepsilon \tau_2}$$

$$\frac{\Delta; \alpha; \cdot; \Gamma \vdash h : \tau_h \mathbin{@} \tau_o \;/\; \varepsilon_o \qquad \Delta, \alpha; \Gamma, x : \tau_i \vdash e_r : \tau_o \;/\; \varepsilon_o}{\Delta; \Gamma \vdash \mathtt{handler}\ h\ \mathtt{ret}\ x \Rightarrow e_r : \mathsf{H}\alpha.\, \tau_h \mathbin{@} \tau_i \;/\; \varepsilon_o \Rightarrow \tau_o \;/\; \varepsilon_o} \qquad \frac{\Delta; \Gamma \vdash v : \tau \qquad \tau <: \tau'}{\Delta; \Gamma \vdash v : \tau'}$$

**Typing handlers.** $\boxed{\Delta_1; \alpha; \Delta_2; \Gamma \vdash h : \tau \mathbin{@} \tau \;/\; \varepsilon}$

$$\frac{\Delta_1, \alpha, \Delta_2; \Gamma, x : \tau_1, r : \tau_2 \to_\varepsilon \tau \vdash e : \tau \;/\; \varepsilon}{\Delta_1; \alpha; \Delta_2; \Gamma \vdash \mathtt{eff}\ x/r.\, e : \tau_1 \to_\alpha \tau_2 \mathbin{@} \tau \;/\; \varepsilon} \qquad \frac{\Delta_1; \alpha; \Delta_2; \Gamma \vdash h : \tau_1 \mathbin{@} \tau \;/\; \varepsilon \qquad \tau_1 <: \tau_2}{\Delta_1; \alpha; \Delta_2; \Gamma \vdash h : \tau_2 \mathbin{@} \tau \;/\; \varepsilon}$$

Figure 3.2: Typing rules of the minimal calculus

## 3.3 Semantics

The reduction semantics of the calculus are given in Figure 3.3. The call-by-value beta reduction rule for function application is standard. In order to give semantics to the `handle` construct, we add two runtime expressions: the labeled $\mathsf{shift}_0$ ($\mathtt{shift}_0^l\ x.\, e$) and $\mathsf{reset}_0$ ($\langle e \mid x.\, e \rangle^l$) delimited control operators [Danvy and Filinski 1989; Ikemori *et al.* 2023; Shan 2007], which are commonly used to implement lexical effect handlers [Brachthäuser *et al.* 2020a,b]. The two rules for delimited control are fairly standard. For the evaluation contexts $E$, we use the notation $E^L$ to specify that there is no $\mathsf{reset}_0$ with a label $l \in L$ around the hole. A $\mathtt{shift}_0^l\ r.\, e$ is evaluated by capturing the context up to (and including) the nearest $\mathsf{reset}_0$ with a matching label, and replacing it with the expression $e$. In case the expression under the $\mathsf{reset}_0$ evaluates to a value, the delimiter is removed and the return expression is evaluated with that value.

A $\mathtt{handle}\ x = h\ \mathtt{ret}\ y \Rightarrow e'\ \mathtt{in}\ e$ expression is evaluated by generating a fresh label $l$,

$$\text{Expr} \ni e ::= \ldots \mid \mathtt{shift}_0^l\ x.\, e \mid \langle e \mid x.\, e \rangle^l \mid \ldots \qquad \text{(expressions)}$$
$$\text{ECtx} \ni E ::= \square \mid E\ e \mid v\ E \mid \langle E \mid x.\, e \rangle^l \mid \ldots \qquad \text{(evaluation contexts)}$$

$$\begin{aligned}
E[(\lambda x.\, e)\ v] &\longrightarrow E[e\{v/x\}] \\
E[\mathtt{handle}\ x = h\ \mathtt{ret}\ y \Rightarrow e'\ \mathtt{in}\ e] &\longrightarrow E[\langle e\{|h|^l/x\} \mid y.\, e' \rangle^l] && l\text{—fresh} \\
E[\langle E_r^{\{l\}}[\mathtt{shift}_0^l\ r.\, e] \mid x.\, e' \rangle^l] &\longrightarrow E[e\{\lambda y.\, \langle E_r^{\{l\}}[y] \mid x.\, e' \rangle^l / r\}] \\
E[\langle v \mid x.\, e \rangle^l] &\longrightarrow E[e\{v/x\}] \\
|\mathtt{eff}\ x/r.\, e|^l &\triangleq \lambda x.\, \mathtt{shift}_0^l\ r.\, e
\end{aligned}$$

Figure 3.3: Runtime constructs and operational semantics of the minimal calculus

setting up a $\mathsf{reset}_0$ with this label and the return part of the handler, and transforming the $h$ part of the handler value into a capability using the auxiliary $|\cdot|^l$ operation. For now, we only have to define $|\cdot|^l$ for $\mathtt{eff}\ x/r.\, e$, which is translated into a lambda abstraction that immediately performs $\mathsf{shift}_0$.

# 4 Logical Relations for Effect Capabilities

We will begin with a description of the common components of all the relational models that we will look at. For the interpretation of algebraic effects, we follow the approach proposed by Biernacki *et al.* [2018]. The construction relies on step-indexing [Ahmed 2006; Appel and McAllester 2001]. As in their work, we keep the step indices implicit by working in an intuitionistic logic with the later modality, written $\triangleright$, which forces the step index to advance [Appel *et al.* 2007; Dreyer *et al.* 2011]. Let $\mathbf{UPred}(X)$ denote the step-indexed unary predicates over $X$. We define the spaces of semantic types and effects as follows.

$$\llbracket \mathsf{T} \rrbracket \triangleq \mathbf{UPred}(\mathrm{Val}^2)$$
$$\llbracket \mathsf{E} \rrbracket \triangleq \left\{ F \in \mathbf{UPred}(\mathrm{Expr}^2 \times \mathcal{P}(\mathrm{Label})^2 \times \llbracket \mathsf{T} \rrbracket) \;\middle|\; \bigcup_{(e_1,e_2,L_1,L_2,R)\in F} L_1 \cup L_2 \;\text{ is finite} \right\}$$

In the interest of generality, we will consider binary logical relations. The semantic types are thus the usual relations on pairs of values. The semantic effects relate a pair of expressions that raise the effect, the sets of labels that these expression may be stuck on, and the semantic type of the result that is passed to the resumption. Finally, we impose an additional condition on every semantic effect: the union of all the label sets in the effect is finite. The technical reason to require this property is that it allows us to generate fresh labels as needed in some of the proofs.

In Figure 4.1 we present the denotations of types and effects of the minimal calculus. Since we are dealing with type variables, the denotations work with an environment $\eta \in \llbracket \Delta \rrbracket \triangleq \prod \alpha^\kappa \in \Delta.\, \llbracket \kappa \rrbracket$ (while the type variable scope $\Delta$ is implicit). The interpretation of effects uses the empty set for the pure effect and union of two interpretations for the join of two effects. The interpretation of the unit and arrow types are fairly standard. Due to its

$$\llbracket \alpha^\kappa \rrbracket_\eta \triangleq \eta(\alpha^\kappa) \qquad \llbracket \iota \rrbracket_\eta \triangleq \varnothing$$
$$\llbracket 1 \rrbracket_\eta \triangleq \{(\langle\rangle, \langle\rangle)\} \qquad \llbracket \varepsilon_1 \cdot \varepsilon_2 \rrbracket_\eta \triangleq \llbracket \varepsilon_1 \rrbracket_\eta \cup \llbracket \varepsilon_2 \rrbracket_\eta$$
$$\mathrm{Arr}\ R_1\ R_2\ F \triangleq \{(v_1, v_2) \mid \forall (v_1', v_2') \in R_1.\, (v_1\ v_1', v_2\ v_2') \in \mathcal{E}\ R_2\ F\}$$
$$\llbracket \tau_1 \to_\varepsilon \tau_2 \rrbracket_\eta \triangleq \mathrm{Arr}\ \llbracket \tau_1 \rrbracket_\eta\ \llbracket \tau_2 \rrbracket_\eta\ \llbracket \varepsilon \rrbracket_\eta$$
$$\llbracket \mathsf{H}\alpha.\, \tau_h\ @\ \tau_i\ /\ \varepsilon_i \Rightarrow \tau_o\ /\ \varepsilon_o \rrbracket_\eta \triangleq \{(\texttt{handler}\ h_1\ \texttt{ret}\ x \Rightarrow e_1, \texttt{handler}\ h_2\ \texttt{ret}\ x \Rightarrow e_2) \mid$$
$$((h_1, x.\, e_1), (h_2, x.\, e_2)) \in \mathcal{H}\llbracket \alpha.\, \tau_h\ @\ \tau_i\ /\ \varepsilon_i \Rightarrow \tau_o\ /\ \varepsilon_o \rrbracket_\eta\}$$

Figure 4.1: Interpretation of types and effects

$$
\begin{aligned}
(e_1, e_2) \in \mathcal{E}\ R\ F &\iff \forall (E_1, E_2) \in \mathcal{K}\ R\ F.\ (E_1[e_1], E_2[e_2]) \in \mathrm{Obs} \\
(E_1, E_2) \in \mathcal{K}\ R\ F &\iff (\forall (v_1, v_2) \in R.\ (E_1[v_1], E_2[v_2]) \in \mathrm{Obs}) \wedge \\
&\qquad (\forall (e_1, e_2) \in \mathcal{S}\ R\ F.\ (E_1[e_1], E_2[e_2]) \in \mathrm{Obs}) \\
(E_1^{L_1}[e_1], E_2^{L_2}[e_2]) \in \mathcal{S}\ R\ F &\iff \exists R'.\ (e_1, e_2, L_1, L_2, R') \in F \wedge \\
&\qquad \forall (v_1, v_2) \in \triangleright R'.\ (E_1^{L_1}[v_1], E_2^{L_2}[v_2]) \in \triangleright \mathcal{E}\ R\ F \\
(e_1, e_2) \in \mathrm{Obs} &\iff e_1 = \langle\rangle \wedge e_2 \longrightarrow^* \langle\rangle \vee \exists e_1'.\ e_1 \longrightarrow e_1' \wedge (e_1', e_2) \in \triangleright \mathrm{Obs}
\end{aligned}
$$

Figure 4.2: Biorthogonal closure operators

$$
\begin{aligned}
(\gamma_1, \gamma_2) \in [\![\Gamma]\!]_\eta &\iff \forall (x : \tau) \in \Gamma.\ (\gamma_1(x), \gamma_2(x)) \in [\![\tau]\!]_\eta \\
\Delta; \Gamma \models v_1 \precsim v_2 : \tau &\iff \forall \eta \in [\![\Delta]\!], (\gamma_1, \gamma_2) \in [\![\Gamma]\!]_\eta.\ (\gamma_1^* v_1, \gamma_2^* v_2) \in [\![\tau]\!]_\eta \\
\Delta; \Gamma \models e_1 \precsim e_2 : \tau \ / \ \varepsilon &\iff \forall \eta \in [\![\Delta]\!], (\gamma_1, \gamma_2) \in [\![\Gamma]\!]_\eta.\ (\gamma_1^* e_1, \gamma_2^* e_2) \in \mathcal{E}[\![\tau]\!]_\eta [\![\varepsilon]\!]_\eta \\
\Delta_1; \alpha; \Delta_2; \Gamma \models h_1 \precsim h_2 : \tau_h \ @\ \tau \ / \ \varepsilon & \qquad \textit{(defined in the next three sections)}
\end{aligned}
$$

Figure 4.3: Logical interpretations of typing judgements

utility in a few later definitions, we separately define the relation $\mathrm{Arr}\ R_1\ R_2\ F$, which works on semantic types and effects, and use it to interpret arrow types. This relation makes use of the expression closure $\mathcal{E}\ R_2\ F$, which we will define in a moment. Part of the difficulty of creating a satisfactory model for effect capabilities lies in giving an interpretation to the handler type $\mathsf{H}\alpha.\ \tau_h\ @\ \tau_i \ / \ \varepsilon_i \Rightarrow \tau_o \ / \ \varepsilon_o$. For now, we merely deconstruct the handler value syntax and pass on the components to an auxiliary relation $\mathcal{H}[\![\alpha.\ \tau_h\ @\ \tau_i \ / \ \varepsilon_i \Rightarrow \tau_o \ / \ \varepsilon_o]\!]_\eta$. We will consider several possible definitions of this relation later in this chapter.

The biorthogonal closure operators are defined in Figure 4.2. Following the approach of Biernacki *et al.* [2018], we make use of the stuck expression closure $\mathcal{S}\ R\ F$. Each stuck expression is decomposed into an expression directly raising an effect from $F$, and a context with no reset matching the label of the effect. The contexts, once resumed by plugging in values from the result semantic type, should be semantically well-typed expressions. This last condition is guarded by the later operator to ensure the mutually recursive closure operators are well-defined. However, since the expressions are stuck, at least one computation step will occur before this property is needed. In the expression and context closures we observe the property Obs, which relates two programs $e_1$, $e_2$ if termination of the first implies termination of the second, specialized to just the unit type and stated using step-indexing.

We present logical relations corresponding to each of the typing judgements in Figure 4.3. These allow us to reason about open terms, and to prove compatibility of the relational model with the type system. Term variable environments are interpreted as the standard relations on substitutions, while the relations for values and expressions quantify

over all semantically well-kinded type variable environments and semantically well-typed term variable substitutions. We will delay the definition of the logical relation for handlers, as the design space here requires more discussion. While it is tempting to apply the same strategy as we did for values and expressions, it is not as straightforward. Its closest counterpart among the closed relations is the handler relation $\mathcal{H}[\![\alpha.\, \tau_h \mathbin{@} \tau_i \mathbin{/} \varepsilon_i \Rightarrow \tau_o \mathbin{/} \varepsilon_o]\!]_\eta$, which uses a return clause and looks more like a handler type (with the input and output types and effects) than the handler judgement (with just the delimiter type and effect). As it turns out, some subtle choices in defining this relation have a profound effect on its ability to scale to a variety of handler constructors.

## 4.1 Zhang and Myers' Model

Zhang and Myers [2019] defined logical relations for their $\lambda_{\Updownarrow}$ calculus, which also features effect capabilities. However, the only shape of handler that they allow is comparable to the `eff` construct, and the construction directly exploits this fact by decomposing the handlers and relating the subexpressions. If we additionally restrict handler types such that the abstract effect $\alpha$ can only appear by the arrow in the capability's type[1], we can use Zhang and Myers' approach to create a model of our minimal calculus. Unlike their model, in the definition of $\mathcal{H}$ we need to existentially quantify the semantic type $R_d$ and effect $F_d$ of the delimiter. This is in part due to how subtyping interacts with the handler type's output type and effect, but would also enable us to add the finally clause to handlers fairly easily. In absence of the finally clause, we require that $R_d \subseteq [\![\tau_o]\!]_\eta$ and $[\![\varepsilon_i]\!]_\eta \subseteq F_d \subseteq [\![\varepsilon_o]\!]_\eta$. Here set inclusion is the semantic counterpart of both the subtyping and subeffecting relations. Ultimately, the entire computation with `handle` has to be of type $\tau_o$ and effect $\varepsilon_o$. However, the real delimiter type and effect can be smaller, and are used in the resumption type, where they appear in a negative position of the definition. Due to the negative occurrences we cannot use $\tau_o$ and $\varepsilon_o$ throughout. The rest of the definition of $\mathcal{H}$ is straightforward. First, it quantifies over all semantically well-typed arguments $v_1, v_2$ and resumptions $v_1', v_2'$, and substitutes them into the body of the effectful function. Similarly, the arguments for the return clause are substituted into its body.

$$
\begin{aligned}
&(\texttt{eff}\ x/r.\, e_1, x.\, e_1', \texttt{eff}\ x/r.\, e_2, x.\, e_2') \in \mathcal{H}[\![\alpha.\, \tau_1 \to_\alpha \tau_2 \mathbin{@} \tau_i \mathbin{/} \varepsilon_i \Rightarrow \tau_o \mathbin{/} \varepsilon_o]\!]_\eta \iff \\
&\quad \exists R_d \subseteq [\![\tau_o]\!]_\eta, F_d \subseteq [\![\varepsilon_o]\!]_\eta. \\
&\qquad [\![\varepsilon_i]\!]_\eta \subseteq F_d \wedge \\
&\qquad (\forall (v_1, v_2) \in [\![\tau_1]\!]_\eta. \forall (v_1', v_2') \in \mathsf{Arr}\ [\![\tau_2]\!]_\eta\ R_d\ F_d. \\
&\qquad\quad (e_1\{v_1/x, v_1'/r\}, e_2\{v_2/x, v_2'/r\}) \in \mathcal{E}\ R_d\ F_d) \wedge \\
&\qquad (\forall (v_1, v_2) \in [\![\tau_i]\!]_\eta. (e_1'\{v_1/x\}, e_2'\{v_2/x\}) \in \mathcal{E}\ R_d\ F_d)
\end{aligned}
$$

We can also see what benefit we get from restricting where the effect $\alpha$ can appear. Had it been allowed to occur in any of $\tau_1$, $\tau_2$, $\tau_i$, $\varepsilon_i$, $\tau_o$ or $\varepsilon_o$, we would be forced to extend the environment $\eta$ with some interpretation for this variable.

[1] The abstract effect continues to be implicitly added to the input effect in the typing rule for `handle`.

For the semantic judgement for open handlers, we take a very similar approach, albeit this time we do not have to deal with the return clause. As long as we stick to the restrictions on the use of $\alpha$, we are safe to consider type variables from just $\Delta_1, \Delta_2$ for the environment $\eta$.

$$\begin{aligned}
&\Delta_1; \alpha; \Delta_2; \Gamma \models \texttt{eff}\ x/r.\, e_1 \precsim \texttt{eff}\ x/r.\, e_2 : \tau_1 \to_\alpha \tau_2 \ @\ \tau \ /\ \varepsilon \iff \\
&\quad \forall \eta \in [\![\Delta_1, \Delta_2]\!].\, \forall (\gamma_1, \gamma_2) \in [\![\Gamma]\!]_\eta. \\
&\qquad \forall (v_1, v_2) \in [\![\tau_1]\!]_\eta.\, \forall (v_1', v_2') \in [\![\tau_2 \to_\varepsilon \tau]\!]_\eta. \\
&\qquad\quad ((\gamma_1^*\, e_1)\{v_1/x, v_1'/r\}, (\gamma_2^*\, e_2)\{v_2/x, v_2'/r\}) \in \mathcal{E}[\![\tau]\!]_\eta [\![\varepsilon]\!]_\eta
\end{aligned}$$

With all the definitions in place, we can prove the fundamental property of the logical relation, and conclude type safety of the type system of the minimal calculus with some extra restrictions.

**Theorem 1 (Fundamental Property)** *If* $\Delta; \Gamma \vdash e : \tau \ /\ \varepsilon$ *then* $\Delta; \Gamma \models e \precsim e : \tau \ /\ \varepsilon$.

**Theorem 2 (Type Safety)** *For each expression* $e$ *such that* $\cdot; \cdot \vdash e : 1 \ /\ \iota$ *if* $e \longrightarrow^* e'$ *then the expression* $e'$ *is a value or can do another reduction step* $e' \longrightarrow e''$ *to some expression* $e''$.

Finally, for the restricted calculus, we can prove that the logical relation is sound with respect to contextual approximation $\Delta; \Gamma \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau \ /\ \varepsilon$ defined in the standard way (see [Ahmed 2006] or the Coq formalization).

**Theorem 3 (Soundness w.r.t. Contextual Approximation)** *If* $\Delta; \Gamma \models e_1 \precsim e_2 : \tau \ /\ \varepsilon$ *then* $\Delta; \Gamma \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau \ /\ \varepsilon$.

## 4.2 Existential Semantic Effect Model

Our goal for the logical relations is to accommodate the addition of many different shapes of handlers, such as pairs for multiple operations, lambda abstractions for additional parameters, type abstractions for polymorphic operations, and so on. However, instead of modifying the interpretation of handlers each time a new construct is added, we would prefer a model that is inherently robust to such changes in the calculus. Therefore, an intensional definition that deconstructs the handler is not suitable. We are seeking an extensional definition, which describes how the handler is used, and not how it is constructed.

In this and the following section, we will propose two different approaches to defining the handler interpretation $(h_1, x.\, e_1', h_2, x.\, e_2') \in \mathcal{H}[\![\alpha.\, \tau_h \ @\ \tau_i \ /\ \varepsilon_i \Rightarrow \tau_o \ /\ \varepsilon_o]\!]_\eta$ and the semantic judgement $\Delta_1; \alpha; \Delta_2; \Gamma \models h_1 \precsim h_2 : \tau_h \ @\ \tau \ /\ \varepsilon$, both of which will rely on relating the capabilities $|h_1|^{l_1}$ and $|h_2|^{l_2}$, for any pair of labels $l_1, l_2$. To do that, we will need to interpret the type $\tau_h$, which most likely contains $\alpha$. This means that the restrictions used in the previous section no longer let us avoid producing a semantic effect for the handlers.

The possible effect-raising expressions and resumption argument types for the handlers' semantic effect depend on the handlers' shape. For $\texttt{eff}\ x/r.\, e$, for which the capability is $\lambda x.\, \texttt{shift}_0^l\ r.\, e$, the expressions will look like $\texttt{shift}_0^l\ r.\, e\{v/x\}$, where $v$ has the semantic type of the function's argument. Indeed, this kind definition would have allowed us to lift

the various restrictions in the presented model of the minimal calculus. However, now that we are about to enrich the calculus, that kind of model will not scale to the desired extensions.

As a first step towards modeling the full-fledged programming language from Chapter 2, we want to support effects with multiple operations through pairs of handlers. We need to extend the calculus with standard value pairs $\langle v_1, v_2 \rangle$, projections, and product types $\tau_1 \times \tau_2$. The typing rule for the handler pair $\langle h_1, h_2 \rangle$ is given below.

$$\frac{\Delta_1; \alpha; \Delta_2; \Gamma \vdash h_1 : \tau_1 \ @ \ \tau \ / \ \varepsilon \qquad \Delta_1; \alpha; \Delta_2; \Gamma \vdash h_2 : \tau_2 \ @ \ \tau \ / \ \varepsilon}{\Delta_1; \alpha; \Delta_2; \Gamma \vdash \langle h_1, h_2 \rangle : \tau_1 \times \tau_2 \ @ \ \tau \ / \ \varepsilon}$$

Unsurprisingly, the capability for a handler pair is defined as a value pair of the capabilities of the components: $|\langle h_1, h_2 \rangle|^l \triangleq \langle |h_1|^l, |h_2|^l \rangle$.

Now we turn our attention back to finding an appropriate semantic effect for a given pair of handlers. The effect-raising expressions for a handler like $\langle \texttt{eff}\ x/r.\, e_1, \texttt{eff}\ x/r.\, e_2 \rangle$ can can come from the $\text{shift}_0$ from either of the two effectful functions, and still more complicated shapes of handlers are now possible. In the spirit of making our definitions more extensional, instead of using a specific semantic effect, we can specify the behavior that we require of the effect. For this purpose, below we define a relation $F \in \mathcal{F}\ l_1\ l_2\ R_d\ F_d$, which tells us that $F$ is a suitable semantic effect for handlers with the given pair of labels $l_1, l_2$ and delimiter type $R_d$ and effect $F_d$.

$$\begin{array}{l}
F \in \mathcal{F}\ l_1\ l_2\ R_d\ F_d \iff \\
\qquad \textit{(For every pair of stuck expressions } E_1^{L_1}[e_1], E_2^{L_2}[e_2] \textit{ built from } F\textit{...)} \\
\quad \forall (e_1, e_2, L_1, L_2, R) \in F.\, \forall E_1^{L_1}, E_2^{L_2}. \\
\qquad \textit{(...and return clauses } e_1', e_2' \textit{ for resets...)} \\
\quad\ \forall e_1', e_2'. \\
\qquad \textit{(...if contexts } E_1^{L_1}, E_2^{L_2} \textit{ under resets for } l_1 \textit{ and } l_2 \textit{ are related...)} \\
\qquad \big(\forall (v_1, v_2) \in \triangleright R.\, (\langle E_1^{L_1}[v_1] \mid x.\, e_1' \rangle^{l_1}, \langle E_2^{L_2}[v_2] \mid x.\, e_2' \rangle^{l_2}) \in \triangleright \mathcal{E}\ R_d\ F_d\big) \Rightarrow \\
\qquad \textit{(...then so are the stuck expressions with the same resets.)} \\
\qquad \big(\langle E_1^{L_1}[e_1] \mid x.\, e_1' \rangle^{l_1}, \langle E_2^{L_2}[e_2] \mid x.\, e_2' \rangle^{l_2}\big) \in \mathcal{E}\ R_d\ F_d
\end{array}$$

Though there is quite a lot going on in this definition, the main idea is that effect-raising expressions from $F$ can be handled by resets on the labels $l_1, l_2$, and the larger delimited expressions built in this way are related using the correct delimiter type and effect. Though this property does not mention the handlers, capabilities, or the capability type, it turns out that it is all we need to know about the effect in the interpretation of the handler type.

The following is the revised definition, which universally quantifies over all labels $l_1, l_2$, and provides the semantic effect $F$ with an existential quantifier. This effect is used thorough to extend the environment $\eta$ with an interpretation for $\alpha$. As previously hinted, this time

we relate the capabilities created from the handlers for the labels $l_1, l_2$.

$$\begin{aligned}
&(h_1, x.\, e_1', h_2, x.\, e_2') \in \mathcal{H}[\![\alpha.\, \tau_h \mathbin{@} \tau_i \mathbin{/} \varepsilon_i \Rightarrow \tau_o \mathbin{/} \varepsilon_o]\!]_\eta \iff \\
&\quad \exists R_d \subseteq [\![\tau_o]\!]_\eta, F_d \subseteq [\![\varepsilon_o]\!]_\eta.\, \forall l_1, l_2.\, \exists F \in \mathcal{F}\ l_1\ l_2\ R_d\ F_d. \\
&\qquad [\![\varepsilon_i]\!]_{\eta[\alpha \mapsto F]} \subseteq F \cup F_d \wedge \\
&\qquad (|h_1|^{l_1}, |h_2|^{l_2}) \in [\![\tau_h]\!]_{\eta[\alpha \mapsto F]} \wedge \\
&\qquad \forall (v_1, v_2) \in [\![\tau_i]\!]_{\eta[\alpha \mapsto F]}.\, (e_1'\{v_1/x\}, e_2'\{v_2/x\}) \in \mathcal{E}\ R_d\ F_d
\end{aligned}$$

The semantic handler judgement also needs to be revised. However, this is not as mechanical a task as one might have hoped. To better understand where various pieces come from, we will now introduce one more handler construct to the calculus: type abstraction. The following typing rule is very similar to how one would define normal type abstraction, but it makes use of the second type variable environment $\Delta_2$ to store the new type variable.

$$\frac{\Delta_1; \alpha; \Delta_2, \beta^\kappa; \Gamma \vdash h : \tau_h \mathbin{@} \tau \mathbin{/} \varepsilon}{\Delta_1; \alpha; \Delta_2; \Gamma \vdash \Lambda h : \forall \beta^\kappa.\, \tau_h \mathbin{@} \tau \mathbin{/} \varepsilon}$$

While the importance of the separation of the two type variable environments is not apparent from the syntactic typing judgement, it plays nicely with the semantic counterpart. The type and effect of the delimiter can make use of variables from $\Delta_1$, but never $\alpha$ or $\Delta_2$. On the other hand, we want to permit $\alpha$ and $\Delta_2$ to be used by $\Gamma$. We can motivate this with yet another handler constructor, the let definition $\texttt{let}\ x = h_1\ \texttt{in}\ h_2$, with its capability given as $|\texttt{let}\ x = h_1\ \texttt{in}\ h_2|^l \triangleq |h_2|^l\{|h_1|^l/x\}$.

$$\frac{\Delta_1; \alpha; \Delta_2; \Gamma \vdash h_1 : \tau_1 \mathbin{@} \tau \mathbin{/} \varepsilon \qquad \Delta_1; \alpha; \Delta_2; \Gamma, x : \tau_1 \vdash h_2 : \tau_2 \mathbin{@} \tau \mathbin{/} \varepsilon}{\Delta_1; \alpha; \Delta_2; \Gamma \vdash \texttt{let}\ x = h_1\ \texttt{in}\ h_2 : \tau_2 \mathbin{@} \tau \mathbin{/} \varepsilon}$$

In the typing rule, the type $\tau_1$, added to the term variable environment in the second premise, should be allowed to contain $\alpha$, since it is the type of the capability for $h_1$.

As our first—and unfortunately insufficient, but only just—attempt, we can try to define the semantic handler judgement as follows.

$$\begin{aligned}
&\Delta_1; \alpha; \Delta_2; \Gamma \models h_1 \precsim h_2 : \tau_h \mathbin{@} \tau \mathbin{/} \varepsilon \iff \\
&\quad \forall \eta_1 \in [\![\Delta_1]\!]. \\
&\quad \forall l_1, l_2.\, \exists F \in \mathcal{F}\ l_1\ l_2\ [\![\tau]\!]_{\eta_1}\ [\![\varepsilon]\!]_{\eta_1}. \\
&\quad \forall \eta_2 \in [\![\Delta_2]\!].\, \forall (\gamma_1, \gamma_2) \in [\![\Gamma]\!]_{\eta_1[\alpha \mapsto F]\eta_2}. \\
&\qquad (|\gamma_1^*\, h_1|^{l_1}, |\gamma_2^*\, h_2|^{l_2}) \in [\![\tau_h]\!]_{\eta_1[\alpha \mapsto F]\eta_2}
\end{aligned}$$

First, we quantify over all environments $\eta_1$ for $\Delta_1$. Then come the arbitrary labels $l_1, l_2$, and the existentially quantified semantic effect $F$, which does not depend on the variables in $\Delta_2$. Where did the existential quantifier come from? While the universally quantified environments are something that is *received* by the handler from its surroundings, the semantic effect is *created* by this particular handler, since it is the handler that determines

the capability and, ultimately, the effect-raising expressions. After the semantic effect $F$, we quantify over the rest of the type variable environment, $\eta_2 \in [\![\Delta_2]\!]$, and the value substitutions $\gamma_1, \gamma_2$, which have access to the full type environment $\eta_1[\alpha \mapsto F]\eta_2$. At last, capabilities are related in the same type variable environment. The alternation of universal and existential quantifiers is unusual, but appears necessary for the fundamental property proof to work. For example, it naturally fits into the proof that the typing rule for type abstraction in handlers holds for the semantic judgement as well. Intuitively, the effect associated with the handler should already exist before the polymorphic effect operation receives its type argument, which is reflected in this definition.

One last modification is needed before the model can support all desired features, such as pairs. The problem with the previous definition is that if we need to use a supereffect $F'$ of $F$, such as we might create by the union of two different semantic effects extracted from the two premises for the handler pair, the assumption about $(\gamma_1, \gamma_2) \in [\![\Gamma]\!]_{\eta_1[\alpha \mapsto F]\eta_2}$ cannot, in general, be proven, and the conclusion about the capabilities is useless in any case. Merely knowing that $F' \supseteq F$ is not helpful, because $\alpha$ can appear in both positive and negative positions in $\Gamma$ and $\tau_h$. The key insight to solve this issue is that the effect $F$ describes only one small part of a potentially more complex handler. Thus, what we really need is for the definition to account for all supereffects of $F$ to allow handlers to be composed.

$$
\begin{aligned}
&\Delta_1; \alpha; \Delta_2; \Gamma \models h_1 \precsim h_2 : \tau_h \mathbin{@} \tau \mathbin{/} \varepsilon \iff \\
&\quad \forall \eta_1 \in [\![\Delta_1]\!]. \\
&\quad \forall l_1, l_2. \exists F \in \mathcal{F}\ l_1\ l_2\ [\![\tau]\!]_{\eta_1}\ [\![\varepsilon]\!]_{\eta_1}. \\
&\quad \boxed{\forall F' \supseteq F.} \\
&\quad \forall \eta_2 \in [\![\Delta_2]\!]. \forall (\gamma_1, \gamma_2) \in [\![\Gamma]\!]_{\eta_1[\alpha \mapsto \boxed{F'}]\eta_2}. \\
&\qquad (|\gamma_1^*\, h_1|^{l_1}, |\gamma_2^*\, h_2|^{l_2}) \in [\![\tau_h]\!]_{\eta_1[\alpha \mapsto \boxed{F'}]\eta_2}
\end{aligned}
$$

It may seem unintuitive that we never had to worry about supereffects in the definition of the $\mathcal{H}[\![\alpha.\, \tau_h \mathbin{@} \tau_i \mathbin{/} \varepsilon_i \Rightarrow \tau_o \mathbin{/} \varepsilon_o]\!]_\eta$ relation. The reason for that is that after we turn a handler into a handler value, we can no longer compose it with other handlers to make a single more complex handler.

With the definitions described in this chapter, along with various more standard value interpretations for types like the product, sum, and universal and existential quantifiers, we can prove Theorem 1, Theorem 2 and Theorem 3 for a calculus with handler constructors such as lambda abstractions, recursive functions, type abstractions, pairs, the left and right injection of the sum type, existential packing, and let. The details pertaining to all these constructs can be found in the accompanying Coq formalization.

## 4.3 Maximal Semantic Effect Model

In the previous section, we avoided defining any specific semantic effect in the handler interpretations by hiding it with existential quantification and a simple property to ensure the effect is good. Since the property relied only on the labels and the type and effect of

the delimiter, and not the handlers, it seems like there might be a concrete one-size-fits-all effect that works for every handler that expects a particular delimiter, and it is indeed the case. First of all, we can observe that all expressions that directly raise an effect are of the form $\mathtt{shift}_0^l\ r.\, e$, no matter how complex the capability. We define the maximal effect $F_{\max}^{l_1,l_2}\ R_d\ F_d$ as the effect containing all pairs of expressions of this shape such that $e_1\{v_1/r\}$ and $e_2\{v_2/r\}$ are related for related resumptions $v_1$ and $v_2$, given a particular resumption argument type.

$$
\begin{aligned}
&(\mathtt{shift}_0^{l_1}\ r.\, e_1, \mathtt{shift}_0^{l_2}\ r.\, e_2, \{l_1\}, \{l_2\}, R) \in F_{\max}^{l_1,l_2}\ R_d\ F_d \iff \\
&\quad \forall (v_1, v_2) \in \mathsf{Arr}\ R\ R_d\ F_d.\ (e_1\{v_1/r\}, e_2\{v_2/r\}) \in \mathcal{E}\ R_d\ F_d
\end{aligned}
$$

As it happens, we can replace the existential quantification in the handler type interpretation by directly using the maximal effect. The definition can remain almost identical.

$$
\begin{aligned}
&(h_1, x.\, e_1', h_2, x.\, e_2') \in \mathcal{H}[\![\alpha.\, \tau_h\ @\ \tau_i\ /\ \varepsilon_i \Rightarrow \tau_o\ /\ \varepsilon_o]\!]_\eta \iff \\
&\quad \exists R_d \subseteq [\![\tau_o]\!]_\eta, F_d \subseteq [\![\varepsilon_o]\!]_\eta.\ \forall l_1, l_2. \\
&\qquad [\![\varepsilon_i]\!]_{\eta[\alpha \mapsto F_m]} \subseteq F_m \cup F_d \wedge \\
&\qquad (|h_1|^{l_1}, |h_2|^{l_2}) \in [\![\tau_h]\!]_{\eta[\alpha \mapsto F_m]} \wedge \\
&\qquad \forall (v_1, v_2) \in [\![\tau_i]\!]_{\eta[\alpha \mapsto F_m]}.\ (e_1'\{v_1/x\}, e_2'\{v_2/x\}) \in \mathcal{E}\ R_d\ F_d \\
&\quad \text{where } F_m = F_{\max}^{l_1,l_2}\ R_d\ F_d
\end{aligned}
$$

The semantic handler judgement can similarly be modified to use the maximal effect. In this case, we can make two observation. Firstly, all the trouble we went to in order to deal with supereffects is no longer necessary. The handlers that we compose all have the same delimiter type and effect, so the semantic effect is always the same. Secondly, the separation of $\Delta_1$ and $\Delta_2$ is a lot less significant once there is no quantifier alternation, and could be removed altogether. Nonetheless we keep it here for the sake of hygiene, as $\tau$ and $\varepsilon$ cannot use $\Delta_2$. The following is the complete definition.

$$
\begin{aligned}
&\Delta_1; \alpha; \Delta_2; \Gamma \models h_1 \precsim h_2 : \tau_h\ @\ \tau\ /\ \varepsilon \iff \\
&\quad \forall l_1, l_2. \\
&\quad \forall \eta_1 \in [\![\Delta_1]\!].\ \forall \eta_2 \in [\![\Delta_2]\!].\ \forall (\gamma_1, \gamma_2) \in [\![\Gamma]\!]_{\eta_1[\alpha \mapsto F_m]\eta_2}. \\
&\qquad (|\gamma_1^*\, h_1|^{l_1}, |\gamma_2^*\, h_2|^{l_2}) \in [\![\tau_h]\!]_{\eta_1[\alpha \mapsto F_m]\eta_2} \\
&\quad \text{where } F_m = F_{\max}^{l_1,l_2}\ [\![\tau]\!]_\eta\ [\![\varepsilon]\!]_\eta
\end{aligned}
$$

As with the existential effect model, we can use the maximal effect model to prove Theorem 1, Theorem 2 and Theorem 3 for the calculus with all the considered extensions.

# 5 Conclusion and Future Work

We proposed not one, but two constructions of logical relations that can scale to a variety of handler definitions that are useful in the practice of programming. The existential model, introduced in Section 4.2, gave us insight into a sufficient condition on the handler's semantic effect and the difference between type variables bound before and after the distinguished effect variable of a handler. It also elegantly hides information about the semantic effect by the use of the existential quantifier. On the other hand, the model from Section 4.3 based on the maximal semantic effect shows us that a concrete semantic effect can be given for all handlers by a coarse approximation from above. This avoids the subtleties of the existential model and allows for some simplifications in the construction. The inclusion of all well-typed $\mathsf{shift}_0$ expressions in the maximal semantic effect also hints at a connection to some interpretation for the $\mathsf{shift}_0$ and $\mathsf{reset}_0$ operators. In fact, the Coq formalization includes such an interpretation, though its description is out of this work's scope. Ultimately, both variants can be used to prove the fundamental property of the logical relations, type safety of the enriched calculus, and entailment of contextual approximation by the logical relations.

The natural question that arises is how the two models relate to each other. At the time for writing, we cannot offer definitive answers on this topic. On the one hand, one would hope to find that the maximal effect model is contained in the existential model by instantiating the existential quantifier with the maximal effect, which satisfies the required property $\mathcal{F}$. However, the open handler relation in the existential model contains quantification over all supereffects, while the one for the maximal effect does not. Although we named this effect *maximal* to emphasize the intuition behind it, there are clearly semantic effects that are not subeffects of the maximal effect, for example due to relating expressions that are not $\mathsf{shift}_0$. Those effects are possible even if we restricted the supereffects to those satisfying the property $\mathcal{F}$. The situation is no easier for the inclusion of the existential model in the maximal effect model. The effect that was picked in the existential model may also not be a subeffect of the maximal effect. However, even if we could use subeffecting to try to prove the inclusion, we would apparently also have to restrict where the distinguished effect variable appears in the type of the capability and input type of the handler.

If the models are indeed different, it would be interesting to see examples of equivalences that can be proven in only one of the models. A possible candidate for a pair that is contained in the existential model, but not the maximal effect model, are two handlers of the reader effect, one implemented as $\texttt{eff}\ x/r.\, r\ 42$, and the other as a pure function with a handler form that we did not discuss, $\texttt{val}\ \lambda x.\, 42$. The handler constructor $\texttt{val}\ v$ simply uses $v$ as the capability. These two handlers can be shown equivalent using the existential model, but we have not been able to produce a similar proof of the same equivalence in the maximal effect model. The difficulty once again lies in the fact that the maximal effect only contains $\mathsf{shift}_0$ expressions, but in this case one of the implementations is pure.

In the face of these problems, we believe it would be worthwhile to consider a third model, which is a slight variation on the maximal effect model. We could change the maximal effect to be the largest semantic effect that is still in $\mathcal{F}$. A potential future direction is to see how all three models relate to each other.

Despite the unanswered questions about the relationship between the two models proposed in the present work, both of them seem interesting due to the distinct observations required for each construction.

# Bibliography


Amal Ahmed. 2006. "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types". In: *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings* (Lecture Notes in Computer Science). Ed. by Peter Sestoft. Vol. 3924. Springer, 69–83. DOI: 10.1007/11693024_6.

Andrew W. Appel and David McAllester. 2001. "An indexed model of recursive types for foundational proof-carrying code". *TOPLAS*, 23, 5, 657–683. DOI: 10.1145/504709.504712.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. "A very modal model of a modern, major, general type system". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, 109–122. DOI: 10.1145/1190216.1190235.

Patrycja Balik and Piotr Polesiuk. 2024. "Logical Relations for Effect Capabilities". To be presented at *HOPE 2024, Milan, Italy, September 2, 2024*.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. "Abstracting algebraic effects". *PACMPL*, 3, POPL, 6:1–6:28. DOI: 10.1145/3290319.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. "Binders by day, labels by night: effect instances via lexically scoped handlers". *PACMPL*, 4, POPL, 48:1–48:29. DOI: 10.1145/3371116.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. "Handle with care: relational interpretation of algebraic effects and handlers". *PACMPL*, 2, POPL, 8:1–8:30. DOI: 10.1145/3158096.

Jonathan I. Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020a. "Effects as capabilities: effect handlers and lightweight effect polymorphism". *PACMPL*, 4, OOPSLA, 126:1–126:30. DOI: 10.1145/3428194.

Jonathan I. Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020b. "Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala". *JFP*, 30, e8. DOI: 10.1017/S0956796820000027.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. "Doo bee doo bee doo". *JFP*, 30, e9. DOI: 10.1017/S0956796820000039.

Olivier Danvy and Andrzej Filinski. 1989. *A Functional Abstraction of Typed Contexts*. Tech. rep.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. "Logical Step-Indexed Logical Relations". *LMCS*, 7, 2. DOI: 10.2168/LMCS-7(2:16)2011.

Daniel Hillerström and Sam Lindley. 2016. "Liberating effects with rows and handlers". In: *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*. Ed. by James Chapman and Wouter Swierstra. ACM, 15–27. DOI: 10.1145/2976022.2976033.

Kazuki Ikemori, Youyou Cong, and Hidehiko Masuhara. 2023. "Typed Equivalence of Labeled Effect Handlers and Labeled Delimited Control Operators". In: *International Symposium on Principles and Practice of Declarative Programming, PPDP 2023, Lisboa, Portugal, October 22–23, 2023*. Ed. by Santiago Escobar and Vasco T. Vasconcelos. ACM, 4:1–4:13. DOI: 10.1145/3610612.3610616.

Daan Leijen. 2014. "Koka: Programming with Row Polymorphic Effect Types". In: *Proceedings of 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, April 12, 2014* (EPTCS). Ed. by Paul B. Levy and Neel Krishnaswami. Vol. 153, 100–126. DOI: 10.4204/EPTCS.153.8.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. "Do be do be do". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 500–514. DOI: 10.1145/3009837.3009897.

Gordon D. Plotkin and Matija Pretnar. 2013. "Handling Algebraic Effects". *LMCS*, 9, 4. DOI: 10.2168/LMCS-9(4:23)2013.

Piotr Polesiuk. 2017. "IxFree: Step-Indexed Logical Relations in Coq". In: *CoqPL 2017, Paris, France, January 21, 2017*.

Piotr Polesiuk and Filip Sieczkowski. 2024. "Functorial Syntax for All". In: *CoqPL 2024, London, United Kingdom, January 20, 2024*.

Chung-chieh Shan. 2007. "A static simulation of dynamic delimited control". *High. Order Symb. Comput.*, 20, 4, 371–401. DOI: 10.1007/s10990-007-9010-4.

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. "Retrofitting effect handlers onto OCaml". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20–25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 206–221. DOI: 10.1145/3453483.3454039.

Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. "First-class names for effect handlers". *PACMPL*, 6, OOPSLA2, 30–59. DOI: 10.1145/3563289.

Yizhou Zhang and Andrew C. Myers. 2019. "Abstraction-safe effect handlers via tunneling". *PACMPL*, 3, POPL, 5:1–5:29. DOI: 10.1145/3290318.